

```

////////////////////////////////////
//
// Title      : ColorLearn.c
// Author     : Siming Lin
// Date      : 
// Copyright  : National Instruments . All Rights Reserved.
// Access    : Company Confidential
// Purpose   : Implements the ColorHistogram function.
//
////////////////////////////////////

//=====
// Include Directives
//=====
#include PRECOMP
#include "Analysis/Color/CommonColor.h"
#include <stdlib.h>
#include <math.h>

////////////////////////////////////
//
// ColorLearn
//
// Description:
//     Extract the color feature from the input color image.
//     Feature:
//     the estimated color spectrum
// Parameters:
//     pSrcRef - source image
//     pCoordPtr - optional rectangle
//     pNbColor - level of color representaion: 0, 1, and 2.
//     pSatThreshold - Threshold for distinguishing two colors with the same hue value
//
//     pColSpectrumRef - calculated color spectrum feature
//
// Return Value:
//     Error code
//
////////////////////////////////////

// algorithms for color feature extraction

enum ColorFeatureSet{
    kColSpectrum = 0,
    kColSignature,
    kColHisIntersection
};

enum ColorComplexity{
    kComplexityLow = 0,
    kComplexityMedium,
    kComplexityHigh
};

#define BWHueThreshold 10 // this one may need be changed to adapt to light and hue
#define BlkLgtThreshold 10 // need more experiment with IMAQ color board here !
#define BWLgtMiddle 128
#define NbColorBinBase 7
#define OffsetColor 18 // center of the red color for LOW complexty

GRLIBError ColorLearn(const GRImage* pSrcIMRef, const ROI* pRoi, int pNbColor, int
    pSatThreshold, Array1D* pColSpectrumRef) {

```

```

GRImage* lMaskImagePtr = NULL;
MaskPart      lMaskPart;
const Pix8*    lMaskPixPtr;
const Pix8*    lMaskLinePtr;

const GRImage*  lSrcIMPtr;
ConstPixelPtr   lSrcPixPtr;

const PixRGB    *lSrcPixRGBPtr, *lBufSrcRGBPtr;

const PixHSL    *lSrcPixHSLPtr, *lBufSrcHSLPtr;

double          *lColSpectrumA1DWPtr;

int              i, lSizeXLW, lSizeYLW, lSrcLineWidth, lColLW, lLineLW ;

double          lImageAreaDW,  lHueStepDW;

int              lNbColFeature, lNbHueBin, lHueIndex,  lNumColorEntry=0;

long lRHHVal, lGSSVal, lBLVVal;
long lhue, lsat, llgt, lBlkThreshold;
Byte lreplace = 191, offset=0;
long lcoring = 0;
double h,s,l;
long lMskLineWidth;
int lpixelsOutsideMask=0;

GRLIBError      error = ERR_SUCCESS;

// lookup table for function f(x)=exp(-0.025*x), x=0 ~ 200
static double expLookup[] = {
7985,    1.0000, 0.9753, 0.9512, 0.9277, 0.9048, 0.8825, 0.8607, 0.8395, 0.8187, 0.      ✓
6219,    0.7788, 0.7596, 0.7408, 0.7225, 0.7047, 0.6873, 0.6703, 0.6538, 0.6376, 0.      ✓
4843,    0.6065, 0.5916, 0.5769, 0.5627, 0.5488, 0.5353, 0.5220, 0.5092, 0.4966, 0.      ✓
3772,    0.4724, 0.4607, 0.4493, 0.4382, 0.4274, 0.4169, 0.4066, 0.3965, 0.3867, 0.      ✓
2938,    0.3679, 0.3588, 0.3499, 0.3413, 0.3329, 0.3247, 0.3166, 0.3088, 0.3012, 0.      ✓
2288,    0.2865, 0.2794, 0.2725, 0.2658, 0.2592, 0.2528, 0.2466, 0.2405, 0.2346, 0.      ✓
1782,    0.2231, 0.2176, 0.2122, 0.2070, 0.2019, 0.1969, 0.1920, 0.1873, 0.1827, 0.      ✓
1388,    0.1738, 0.1695, 0.1653, 0.1612, 0.1572, 0.1534, 0.1496, 0.1459, 0.1423, 0.      ✓
1081,    0.1353, 0.1320, 0.1287, 0.1256, 0.1225, 0.1194, 0.1165, 0.1136, 0.1108, 0.      ✓
0842,    0.1054, 0.1028, 0.1003, 0.0978, 0.0954, 0.0930, 0.0907, 0.0885, 0.0863, 0.      ✓
0655,    0.0821, 0.0801, 0.0781, 0.0762, 0.0743, 0.0724, 0.0707, 0.0689, 0.0672, 0.      ✓
0510,    0.0639, 0.0623, 0.0608, 0.0593, 0.0578, 0.0564, 0.0550, 0.0537, 0.0523, 0.      ✓
0398,    0.0498, 0.0486, 0.0474, 0.0462, 0.0450, 0.0439, 0.0429, 0.0418, 0.0408, 0.      ✓
0310,    0.0388, 0.0378, 0.0369, 0.0360, 0.0351, 0.0342, 0.0334, 0.0325, 0.0317, 0.      ✓

```

```

0.0302, 0.0295, 0.0287, 0.0280, 0.0273, 0.0266, 0.0260, 0.0253, 0.0247, 0.  ✓
0241,
0.0235, 0.0229, 0.0224, 0.0218, 0.0213, 0.0208, 0.0202, 0.0197, 0.0193, 0.  ✓
0188,
0.0183, 0.0179, 0.0174, 0.0170, 0.0166, 0.0162, 0.0158, 0.0154, 0.0150, 0.  ✓
0146,
0.0143, 0.0139, 0.0136, 0.0132, 0.0129, 0.0126, 0.0123, 0.0120, 0.0117, 0.  ✓
0114,
0.0111, 0.0108, 0.0106, 0.0103, 0.0101, 0.0098, 0.0096, 0.0093, 0.0091, 0.  ✓
0089,
0.0087, 0.0084, 0.0082, 0.0080, 0.0078, 0.0076, 0.0074, 0.0073, 0.0071, 0.  ✓
0069,
0.0067, 0.0066, 0.0064, 0.0063, 0.0061, 0.0059, 0.0058, 0.0057, 0.0055, 0.  ✓
0054

```

```
};
```

```
// compute the parameters for the color feature lookup table
```

```
switch(pNbColor) {
```

```
case kComplexityLow:
```

```
    lNbHueBin=NbColorBinBase;
```

```
    lHueStepDW=255.0/(double) lNbHueBin;
```

```
    break;
```

```
case kComplexityMedium:
```

```
    lNbHueBin=NbColorBinBase *2;
```

```
    lHueStepDW=255.0/(double) lNbHueBin;
```

```
break;
```

```
case kComplexityHigh:
```

```
    lNbHueBin=NbColorBinBase *4;
```

```
    lHueStepDW=255.0/(double) lNbHueBin;
```

```
    break;
```

```
default:
```

```
    lNbHueBin=NbColorBinBase;
```

```
    lHueStepDW=255.0/(double) lNbHueBin;
```

```
    break;
```

```
}
```

```
lNbColFeature= lNbHueBin*2 +2;
```

```
// resize the color feature array
```

```
if (error = ResizeArray1D(pColSpectrumRef, sizeof(double), lNbColFeature))
```

```
{
```

```
    goto END;
```

```
}
```

```
// Get the color array pointer
```

```
if (error = GetArray1DPtr(pColSpectrumRef, &lColSpectrumA1DWPtr))
```

```
{
    goto END;
}

/*Initilization */

for (i=0; i<lNbColFeature; i++) {
    lColSpectrumA1DWPtr[i]=0;
}

// get the pointer to the input image structure
lSrcIMPtr = pSrcIMRef;
lSrcLineWidth = lSrcIMPtr->lineWidth;

// if ROI input connected, calculate the color feature based on the ROI image
if( pRoi->numContours>0) {

    if ( error= NewImage( &lMaskImagePtr, IMAGE_U8, 4, 4, 0) ) {
        goto END;
    }

    if (error = ROIToMask(lMaskImagePtr, NULL, pRoi, 1, &lpixelsOutsideMask) ) {
        goto END;
    };

    lMskLineWidth = lMaskImagePtr->lineWidth;
    CalcMaskPart(pSrcIMRef, lMaskImagePtr, &lMaskPart);

    GetConstImagePixel(pSrcIMRef, lMaskPart.startImageX, lMaskPart.startImageY, &
lSrcPixPtr);

    GetConstImagePixel(lMaskImagePtr, lMaskPart.startMaskX, lMaskPart.startMaskY,
(ConstPixelPtr*)&lMaskLinePtr);

    lSizeXLW = lMaskPart.sizeX;
    lSizeY LW = lMaskPart.sizeY;
    lImageAreaDW = 0; /* init the Area */

    // control negative sizes

    if (lSizeXLW < 0)
        lSizeXLW = 0;
    if (lSizeY LW < 0)
        lSizeY LW = 0;

    /* get image pixel base addresses */
    // GetConstImagePixel(pSrcIMRef, lCoord.x1Left, lCoord.y1Top, &lSrcPixPtr);

    switch (lSrcIMPtr->imageType) {

        case IMAGE_RGB32:

            lSrcPixRGBPtr=lSrcPixPtr.PixRGB_Ptr;

            break;
```



```

        if (l1gt < lBlkThreshold ) {
            lNumColorEntry= (lNbColFeature-2);
        }
        else {
            // offset the red color to the center of the first
bin
            lhue = ( lhue + OffsetColor) % 255;
            // clustering non-black-white color feature
            lHueIndex= (double) lhue / lHueStepDW;
            lNumColorEntry= (lsat < pSatThreshold) ? (lHueIndex*2) :
(lHueIndex*2+1);
        }
    }

    lColSpectrumA1DWPptr[lNumColorEntry] += 1.0;
    lImageAreaDW++;
} // end of if (*lMaskPixPtr++)

lBufSrcRGBPtr++;
}

lSrcPixRGBPtr +=lSrcLineWidth;
lMaskLinePtr += lMskLineWidth;
}

// calculate the discrete color spectrume
for( i=0; i< lNbColFeature; i++ ) {
    lColSpectrumA1DWPptr[i] /= lImageAreaDW;
}

break;
case IMAGE_HSL32:
    for (lLineLW = 0; lLineLW < lSizeYLW; lLineLW++) {
        lBufSrcHSLPtr=lSrcPixHSLPtr;
        lMaskPixPtr = lMaskLinePtr;

        for (lColLW = 0; lColLW < lSizeXLW; lColLW++) {
            if (*lMaskPixPtr++) {
                // initialization

```

```

        lhue=lBufSrcHSLPtr->Byte_HSL.H;
        lsat=lBufSrcHSLPtr->Byte_HSL.S;
        llgt=lBufSrcHSLPtr->Byte_HSL.L;

        lNumColorEntry=0;

        if( lsat< BWHueThreshold ) {

            lNumColorEntry=(llgt < BWLgtMiddle ) ? (lNbColFeature-2):
first bin (lNbColFeature-1);

        }

        else {

            // Threshold Black color using Nobelist's formular

            // lBlkThreshold= (long) ((128 - BlkLgtThreshold) *exp(-
0.025*(lsat - BWHueThreshold)) + BlkLgtThreshold );

            if(lsat < 200 )
                lBlkThreshold= (long) ((128 - BlkLgtThreshold) *
expLookup[lsat - BWHueThreshold] + BlkLgtThreshold );
            else
                lBlkThreshold=BlkLgtThreshold;

            if (llgt < lBlkThreshold ) {

                lNumColorEntry= (lNbColFeature-2);

            }

            else {

                // offset the red color to the center of the
first bin

                lhue = ( lhue + OffsetColor) % 255;

                // clustering non-black-white color feature

                lHueIndex= (double) lhue / lHueStepDW;

                lNumColorEntry= (lsat < pSatThreshold) ?
(lHueIndex*2) : (lHueIndex*2+1);

            }

        }

        lColSpectrumA1DWPptr[lNumColorEntry] += 1.0;

        lImageAreaDW++;

    } // end of if (*lMaskPixPtr++)

    lBufSrcHSLPtr++;

}

lSrcPixHSLPtr+=lSrcLineWidth;
lMaskLinePtr += lMskLineWidth;

```

```
    }

    // calculate the discrete color spectrum
    for( i=0; i< lNbColFeature; i++ ) {
        lColSpectrumAlDWPtr[i] /= lImageAreaDW;
    }

    break;

} // end of switch (lSrcIMPtr->imageType)

} // end of ROI case

else { // ROI not connected

    GetConstImagePixel(pSrcIMRef,0,0, &lSrcPixPtr);

    lSizeXLW = lSrcIMPtr->xRes;
    lSizeYLW = lSrcIMPtr->yRes;

    lImageAreaDW = (double) lSizeXLW * lSizeYLW;

    switch (lSrcIMPtr->imageType) {

        case IMAGE_RGB32:

            lSrcPixRGBPtr=lSrcPixPtr.PixRGB_Ptr;

            break;

        case IMAGE_HSL32:

            lSrcPixHSLPtr=lSrcPixPtr.PixHSL_Ptr;

            break;

        default:
            error = ERR_BADIMAGETYPE;
            goto END;

    }

    // main switch

    switch (lSrcIMPtr->imageType) {

        case IMAGE_RGB32:

            for (lLineLW = 0; lLineLW < lSizeYLW; lLineLW++) {

                lBufSrcRGBPtr=lSrcPixRGBPtr;

                for (lColLW = 0; lColLW < lSizeXLW; lColLW++) {

                    // initializatin
```



```

        lNumColorEntry=0;

        // convert the RGB to HSL that corresponds to IMAQ color board

        lRHHVal = lBufSrcRGBPtr->Byte_RGB.R ;
        lGSSVal = lBufSrcRGBPtr->Byte_RGB.G ;
        lBLVVal = lBufSrcRGBPtr->Byte_RGB.B ;

        CCONV_HSI(lRHHVal,lGSSVal,lBLVVal,lcoring,lreplace,l,s,h)
        lhue = (Byte) (h +0.5 + offset) % 256;
        llgt = (Byte) (l+0.5);
        lsat = (Byte) (s + 0.5);

        if( lsat< BWHueThreshold ) {

            lNumColorEntry=(llgt < BWLgtMiddle ) ? (lNbColFeature-2):
(lNbColFeature-1);

        }

        else {

            // Threshold Black color using Nobelist's formular

            //lBlkThreshold= (long) ((128 - BlkLgtThreshold) *exp(-0.
025*(lsat - BWHueThreshold)) + BlkLgtThreshold );

            if(lsat < 200 )
                lBlkThreshold= (long) ((128 - BlkLgtThreshold) *
expLookup[lsat - BWHueThreshold] + BlkLgtThreshold );
            else
                lBlkThreshold=BlkLgtThreshold;

            if (llgt < lBlkThreshold ) {

                lNumColorEntry= (lNbColFeature-2);

            }

            else {

                // offset the red color to the center of the first
bin

                lhue = ( lhue + OffsetColor) % 255;

                // clustering non-black-white color feature

                lHueIndex= (double) lhue / lHueStepDW;

                lNumColorEntry= (lsat < pSatThreshold) ? (lHueIndex*2) :
(lHueIndex*2+1);

            }

        }

        lColSpectrumAlDWPTr[lNumColorEntry] += 1.0;

        lBufSrcRGBPtr++;

```

```

    }

    lSrcPixRGBPtr += lSrcLineWidth;

}

// calculate the discrete color spectrume
for( i=0; i< lNbColFeature; i++ ) {
    lColSpectrumAlDWPTr[i] /= lImageAreaDW;

}

break;

case IMAGE_HSL32:
    for( lLineLW = 0; lLineLW < lSizeYLW; lLineLW++ ) {
        lBufSrcHSLPtr=lSrcPixHSLPtr;

        for( lColLW = 0; lColLW < lSizeXLW; lColLW++ ) {

            // initialization

            lhue=lBufSrcHSLPtr->Byte_HSL.H;
            lsat=lBufSrcHSLPtr->Byte_HSL.S;
            llgt=lBufSrcHSLPtr->Byte_HSL.L;

            lNumColorEntry=0;

            if( lsat< BWHueThreshold ) {
                lNumColorEntry=(llgt < BWLgtMiddle ) ? (lNbColFeature-2):
(lNbColFeature-1);
            }

            else {

                // Threshold Black color using Nobelist's formular

                // lBlkThreshold= (long) ((128 - BlkLgtThreshold) *exp(-
0.025*(lsat - BWHueThreshold)) + BlkLgtThreshold );

                if( lsat < 200 )
                    lBlkThreshold= (long) ((128 - BlkLgtThreshold) *
expLookup[lsat - BWHueThreshold] + BlkLgtThreshold );
                else
                    lBlkThreshold=BlkLgtThreshold;

                if ( llgt < lBlkThreshold ) {

                    lNumColorEntry= (lNbColFeature-2);

                }

                else {

                    // offset the red color to the center of the
first bin

```

```
        lhue = ( lhue + OffsetColor) % 255;
        // clustering non-black-white color feature
        lHueIndex= (double) lhue / lHueStepDW;
        lNumColorEntry= (lsat < pSatThreshold) ?
(lHueIndex*2) :(lHueIndex*2+1);
    }
}

    lColSpectrumAlDWPptr[lNumColorEntry] += 1.0;

    lBufSrcHSLPtr++;
}
lSrcPixHSLPtr+=lSrcLineWidth;
}
// calculate the discrete color spectrum
for( i=0; i< lNbColFeature; i++ ) {
    lColSpectrumAlDWPptr[i] /= lImageAreaDW;
}

    break;

} // end of switch (lSrcIMPptr->imageType)

} // end of if( pRoi->numContours>0)

END:
    return error;
}
```